Computing the Length of a Scheme-like List

```
public abstract class AList
{
  ...
  // Returns the number of elements in this AList.
  public abstract int length();
  ...
```

Computing the Length of a Scheme-like List (cont.)

```
public class EmptyList extends AList
{
  ...
  // Returns 0.
  public int length()
  {
    return 0;
  }
  ...
```

Computing the Length of a Scheme-like List (cont.)

```
public class NEList extends AList
{
  ...
  // Returns 1 + the number of elements in _rest.
  public int length()
  {
    return 1 + _rest.length();
  }
  ...
```

An Implementation in Scheme

```
(define (length a-list)
   (cond
      [(empty? a-list)
       0]
      [(cons?  a-list)
       (add1 (length (rest a-list)))]))
```

An `EmptyList` Object vs. `null`

- An `EmptyList` object can perform a computation, e.g.,

```
// Returns 0.
public int length()
{
  return 0;
}
```

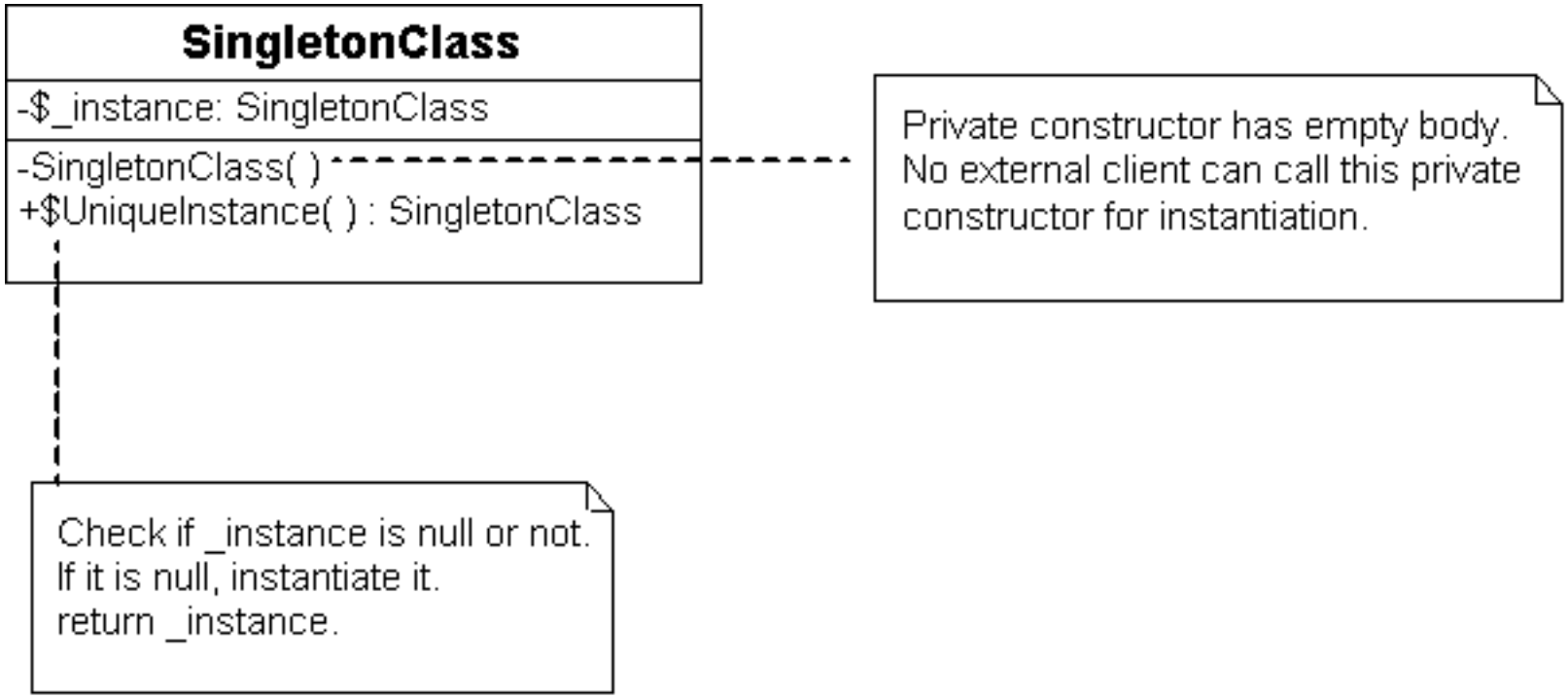
- `null` is not an object. It cannot perform computations.

The Singleton Pattern

- Conceptually, there is only one empty list in the "world".
  - This concept is akin to that of the empty set: there is only one empty set.
- How can we ensure that only one instance of EmptyList can be created throughout the life of a program?
- There is a way to design a class to ensure such uniqueness property. It is called the *Singleton Design Pattern*.

The Singleton Pattern (cont.)

- The following UML diagram describes the pattern:



- Note: The field `_instance` and the method `UniqueInstance()` are of class scope (i.e. static).

The Singleton Pattern (cont.)

- The method `UniqueInstance()` is called a "factory" method as it is used to manufacture an instance, though unique, of the SingletonClass.

- The class SingletonClass is appropriately called a "factory". In this very special case, SingletonClass manufactures its own (unique) instance. (Recall that we saw the *Factory Pattern* in Lab 2.)

One EmptyList Object is Enough

```
public class EmptyList extends AList
{
  private static EmptyList _instance;

  // NOTE: The constructor is private so that no client can
  // instantiate an EmptyList.  I.e., there is one "true"
  // empty list, Singleton, and every list uses it.
  private EmptyList()
  {
  }

  public static EmptyList makeEmptyList()
  {
    if (_instance == null)
      _instance = new EmptyList();
    return _instance;
  }
```

One `EmptyList` Object is Enough

```
public class EmptyList extends AList
{
  public final static EmptyList Singleton = new EmptyList();

  /**
   * NOTE: The constructor is private so that no client can
   * instantiate an EmptyList.  I.e., there is one "true"
   * empty list, Singleton, and every list uses it.
   */
  private EmptyList()
  {
  }
  ...
```

The `final` Modifier

- The `final` modifier prevents
  - the *class*,
  - the *method*, or
  - the *field*

  from being extended or overridden.

- In some sense, `final` is the opposite of `abstract`. Thus, a class/method/field cannot be both `final` and `abstract`.

The `final` Modifier (cont.)

- If a field is declared `final`, then its declaration must include a variable initializer.
  - Example

```
public final static EmptyList Singleton = new EmptyList();
```